Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 301 (2014) 3–19

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Domain-Theoretic Model Of Nominally-Typed Object-Oriented Programming

Moez A. AbdelGawad[1]

Computer Science Department, Rice University
Houston, Texas, USA

Abstract

The majority of contemporary mainstream object-oriented (OO) software is written using nominally-typed OO programming languages. Extant domain-theoretic models of OOP developed to analyze OO type systems miss crucial features of these mainstream OO languages, such as nominality. This paper summarizes the construction of **NOOP** as a domain-theoretic model of OOP that includes nominal information found in nominally-typed mainstream OO software. Inclusion of nominal type information and asserting that type inheritance in statically-typed OO programming languages is an inherently nominal notion allow readily proving that inheritance and subtyping are completely identified in these languages. This conclusion is in full agreement with intuitions of OO developers using these languages, and contrary to the belief that "inheritance is not subtyping", which came from assuming non-nominal structural models of OO type systems. **NOOP**, thus, provides a firmer semantic foundation for analyzing and progressing nominally-typed mainstream OO programming languages.

Keywords: Object-Oriented Programming, Denotational Semantics, Nominative Type Systems, Structural Type Systems, **NOOP**, Type Names, Inheritance, Subtyping, OOP, Java, C#

1 Introduction

To evolve and improve the type systems of mainstream object-oriented programming languages such as Java [18], C# [1], C++ [2], Scala [24] and others that utilize class name information in defining object types and OO subtyping, a precise mathematical model of these languages is needed. A precise model of nominally-typed OOP allows accurate reasoning and analysis of these mainstream OO programming languages. Imprecise models, on the other hand, lead to inaccurate conclusions.

An object in class-based OO languages is associated with its class name and the class names of its superclasses, as part of the meaning of the object. Class

[1] Email: moez@cs.rice.edu

http://dx.doi.org/10.1016/j.entcs.2014.01.002

names, in turn, are associated with class contracts.[2] In class-based OOP, two objects with the same structure that have different class name information are different objects. The different class name information inside the two objects imply the two objects could maintain different class contracts, and thus that the objects could behave dissimilarly. The two objects could thus be considered semantically unequal. Further, in nominally-typed class-based OO languages—ones where types and the subtyping relation make use of class names and of the explicitly-specified type inheritance relation between classes— instances of two classes that are not in the class inheritance hierarchy may not be replaced by each other (*i.e.*, are not "assignment-compatible") since they may not offer the degree of (behavioral) substitutability intended by developers of the two classes.

Despite its clear semantic importance, class name information (henceforth, 'nominal information') that is embedded inside objects of many mainstream OO programming languages is not included in the most recognized denotational models of OOP that exist today. Models of OOP that lack nominal information of mainstream OO languages are structural models of OOP, not nominal ones. Structural models of OOP have led PL researchers to make some conclusions about OOP that contradict the intuitions of the majority of mainstream OO developers. For example, the agreement of (type) inheritance, at the syntactic (*i.e.,* program code) level, and subtyping, at the semantic (*i.e.*, program meaning) level, is a fundamental intuition of OO developers using nominally-typed OO languages. However, extant denotational models of OOP led to the inaccurate conclusion that "inheritance is not subtyping" [11].

Inheritance, in class-based mainstream OO languages, is an inherently nominal notion, due to the informal association of class names with *inherited* class contracts. Hence the discrepancy between conclusions regarding inheritance that are based on a structural view of OOP and the intuitions of the majority of mainstream OO developers, who adopt a nominal view of OOP. This discrepancy motivates considering the inclusion of nominal information in mathematical models of OOP.

This paper presents a summary of the construction of a model of OOP, called **NOOP**, that includes nominal information of mainstream OO programming languages. **NOOP** was first presented in [5]. Having a model of OOP that includes nominal information of nominally-typed OOP should enable progress in the design of type systems of current and future mainstream OO languages. Some features of the type systems of these languages (*e.g.*, generics) seem to crucially depend on nominal information. Accurately understanding and analyzing these features, for the purposes of extending the languages or designing new languages that include them, has proven to be hard when using operational models of OOP or using denotational models of OOP that lack full nominal information found in nominally-typed OO languages. As demonstrated in [6], having a nominal domain-theoretic model of OOP should make the analysis of features of these languages that depend on nominal information easier and more accurate. From the point of view of OO soft-

[2] Class contracts are usually expressed, informally, in code documentation. Class contracts are thus (implicitly) encoded in class names.

ware development, having better mainstream OO languages should result in greater productivity and producing higher-quality code.

This paper is organized as follows. Section 2 briefly presents a list of research related to this paper. Section 3 then starts the formal presentation of **NOOP** by presenting a new records domain constructor, called 'rec', that is used in constructing **NOOP**. Section 4 presents class signatures and other related signature constructs, which are syntactic constructs used to embody the nominal information found in nominally-typed OOP. Section 5 presents the construction of **NOOP**, using 'rec' and signature constructs, and then it presents a proof of the identification of inheritance and subtyping in nominally-typed OOP. Section 6 presents the main conclusions we reached based on developing **NOOP**. Section 7 concludes this paper by presenting further research that can be developed based on **NOOP**.

2 Related Research

NOOP is a denotational model of nominally-typed OOP. Dana Scott invented and developed, with others including Gordon Plotkin, the fields of domain theory and denotational semantics (*e.g.*, see [29,32,30,25,12,19,16]). The development of denotational semantics has been motivated by researching the semantics of functional programming languages such as Lisp [21,22] and ML [17,23].

Research on the semantics of OOP has taken place subsequently. Luca Cardelli built the first widely-known denotational model of OOP [9,10]. Cardelli's model was a structural model that lacked nominal information. William Cook and his colleagues built on Cardelli's work to separate the notions of inheritance and subtyping [13,15,14]. [3] Later, Kim Bruce [8] and Anthony Simons [31] promoted Cardelli and Cook's structural view of OOP, and promoted conclusions based on this view.

Martin Abadi, with Luca Cardelli, later presented operational models of OOP [3,4]. These models also had a structural view of OOP. Operational models with a nominal view of OOP got later developed. In their seminal work, Atsushi Igarashi, Benjamin Pierce, and Philip Wadler presented Featherweight Java [20] (FJ) as an operational model of a nominally-typed OO language. Even though not the first operational model of nominally-typed OOP, FJ is the most widely known operational model of (a tiny core subset of) a nominally-typed OO language, namely Java. It is worthy to mention that **NOOP**—as a more foundational domain-theoretic model of nominally-typed OO languages (including Java)—provides a denotational justification for the inclusion of nominal information in Featherweight Java.

Other research that is similar to one presented here, but that had different research interests and goals, is that of Reus and Streicher [27,28,26]. In [26], an untyped denotational model of class-based OOP is developed. Type information is largely ignored in this work (object methods and fields have no type signatures) and some nominal information is included with objects only to analyze OO dynamic dispatch. The model of [26] was developed to analyze mutation and imperative

[3] A detailed discussion of Cardelli's and Cook's work, and a comparison of **NOOP** to their work, is presented in [6].

features of OO languages and for developing specifications of OO software and the verification of its properties. Analyzing the differences between structurally-typed and nominally-typed OO type systems was not a goal of Reus and Streicher's research, and in their work the identification of inheritance and subtyping was, again (as in FJ), assumed rather than proven as a consequence of nominality and nominal typing.

3 'Rec' ($\multimap$), A New Records Domain Constructor

To construct **NOOP** we introduce a new domain constructor. In addition to **NOOP** including nominal information of mainstream OOP, **NOOP** models records as *tagged finite functions* rather than infinite functions, as another improvement over extant domain-theoretic models of OOP (particularly that of Cardelli [9,10] and other models built directly on top of it).

Due to the finiteness of the shape of an object (the shape of an object is the set of names/labels of its fields and methods), and due to the flatness of the domain of labels when labels are formulated as members of a computational domain, modeling objects in **NOOP** motivates defining a new domain constructor that is similar to but somewhat different from conventional functional domain constructors. This domain constructor, $\multimap$, called 'rec', constructs tagged finite functions, which we call *record functions*. Record functions are explicitly finite mathematical objects.

A domain $\mathcal{R} = \mathcal{L} \multimap \mathcal{D}$ constructed using $\multimap$ is the domain of record functions modeling records with labels from a flat domain $\mathcal{L}$ of labels to an arbitrary domain $\mathcal{D}$ of values. Below we present the records domain constructor, $\multimap$, then we discuss its mathematical properties. The definition of $\multimap$ makes use of the standard definitions from basic domain theory (A summary of these is presented in Appendix A of [5]).

3.1 Record Functions

A record can be viewed as a finite mapping from a set of labels (as member names) to fields or methods. Thus, we model records using explicitly finite record functions. A *record function* is a finite function paired with a tag representing the input domain of the function. The tag of a record function modeling a record represents the set of labels of the record. In agreement with the definition of shapes of objects, we similarly call the set of labels of a record the *shape* of the record. The tag of a record function thus tells the shape of the record.

3.2 Definition of $\multimap$

Let $\mathcal{L}$ be the flat domain containing all record labels plus an extra improper bottom label, $\bot_{\mathcal{L}}$, that makes $\mathcal{L}$ be a domain (all computational domains must have a bottom element). Let $\mathcal{D}$ be an arbitrary domain, with approximation ordering $\sqsubseteq_{\mathcal{D}}$ and bottom element $\bot_{\mathcal{D}}$. Domain $\mathcal{D}$ contains the values that members of records are mapped to.

Let $\Subset$ denote the subdomain relation (see Definition 6.2 in [12]). If we let $\mathcal{L}_f$

range over arbitrary finite subdomains of $\mathcal{L}$ (all subdomains $\mathcal{L}_f$ contain $\bot_{\mathcal{L}}$), then we define the domain $\mathcal{R} = \mathcal{L} \multimap \mathcal{D}$ as the domain of record functions from $\mathcal{L}$ to $\mathcal{D}$, where the universe, $|\mathcal{R}|$, of domain $\mathcal{R}$ is defined by the equation

$$|\mathcal{R}| = \{\bot_{\mathcal{R}}\} \cup \bigcup_{\mathcal{L}_f \Subset \mathcal{L}} R(\mathcal{L}_f, \mathcal{D}) \tag{1}$$

with sets $R(\mathcal{L}_f, \mathcal{D})$ defined as

$$R(\mathcal{L}_f, \mathcal{D}) = \{tag(|\mathcal{L}_f| \setminus \{\bot_{\mathcal{L}}\})\} \times |\mathcal{L}_f \multimap\!\!\to \mathcal{D}| \tag{2}$$

and where *tag* is a function that maps the shape corresponding to a domain $\mathcal{L}_f$ to a unique tag in a countable set of tags (whose exact format does not need to be specified), and where $\mathcal{L}_f \multimap\!\!\to \mathcal{D}$ is the domain of strict continuous functions from $\mathcal{L}_f$ into $\mathcal{D}$. Tags are needed in record functions to ensure that the records domain constructor is a continuous, in fact computable, domain constructor. To illustrate, a record $\mathsf{r} = \{l_1 \mapsto d_1, \cdots, l_k \mapsto d_k\}$ is modeled by a record function $r = (tag(\{l_1, \cdots, l_k\}), \{(\bot_{\mathcal{L}}, \bot_{\mathcal{D}}), (l_1, d_1), \cdots, (l_k, d_k)\})$.

It should be noted that $\multimap$ allows constructing the (unique) record function $(tag(\{\}), \{(\bot_{\mathcal{L}}, \bot_{\mathcal{D}})\})$ that models the empty record (one with an empty set of labels, for which $|\mathcal{L}_f| = \{\bot_{\mathcal{L}}\}$).

The approximation ordering, $\sqsubseteq_{\mathcal{R}}$, over elements of $\mathcal{R}$ is defined as follows. The bottom element $\bot_{\mathcal{R}}$ approximates all elements of the domain $\mathcal{R}$. Non-bottom elements r and r' in $\mathcal{R}$ with unequal tags are unrelated to one another. On the other hand, elements r and r' with the same tag are ordered by their embedded functions (which must be elements of the same domain). Symbolically, for two non-bottom record functions r, r' in $\mathcal{R}$ that are defined over the *same* $\mathcal{L}_f$, where $|\mathcal{L}_f| = \{\bot_{\mathcal{L}}, l_1, \cdots, l_k\}$, if $r = (tag(\{l_1, \cdots, l_k\}), \{(\bot_{\mathcal{L}}, \bot_{\mathcal{D}}), (l_1, d_1), \cdots, (l_k, d_k)\})$ and
$r' = (tag(\{l_1, \cdots, l_k\}), \{(\bot_{\mathcal{L}}, \bot_{\mathcal{D}}), (l_1, d'_1), \cdots, (l_k, d'_k)\})$ where $d_1, \cdots, d_k$ and $d'_1, \cdots, d'_k$ are elements in $\mathcal{D}$, then we define $r \sqsubseteq_{\mathcal{R}} r' \Leftrightarrow \forall_{i \leq k}(d_i \sqsubseteq_{\mathcal{D}} d'_i)$

Theorem 3.1 *Given a flat countable domain of labels $\mathcal{L}$ and an arbitrary domain $\mathcal{D}$, $\mathcal{L} \multimap \mathcal{D}$ is a domain.*

Proof See Appendix B of [5]. □

Having defined the records domain constructor $\multimap$, we now discuss its mathematical properties. Because we will use $\multimap$ to construct domains as least fixed points of functions over domains where the constructed domains need to be subdomains of Scott's universal domain, $\mathcal{U}$, we need to ascertain that $\multimap$ has the domain-theoretic properties needed for it to be used inside these functions. We thus need to prove that $\multimap$ is a continuous function over its input domain $\mathcal{D}$, *i.e.*, that, as a function over domains, $\multimap$ is monotonic with respect to the subdomain relation, $\Subset$, and that it preserves least upper bounds of domains under that relation.

Theorem 3.2 *Domain constructor $\multimap$ is a continuous function on flat domains $\mathcal{L}$ and arbitrary domains $\mathcal{D}$.*

Proof See Appendix B of [5]. □

4 Signatures

In this section we present formal definitions for class signatures and related constructs. Class signatures and other signature constructs are syntactic constructs that capture nominal information found in objects of mainstream OO software. Embedding class signature closures (formally defined below) in objects of **NOOP** makes them nominal objects, thereby making **NOOP** objects more precise models of objects in mainstream OO languages such as Java [18], C# [1], C++ [2], and Scala [24].

Class signatures formalize the informal notion of object interfaces (discussed in [7] and Ch. 2 of [5]). A class signature corresponding to a class in nominally-typed mainstream OOP is a concrete expression of how the instances should be viewed and interacted with by other objects ("the outside world").

To capture nominal information of nominally-typed mainstream OOP, we define three syntactic signature constructs: (1) class signatures, (2) class signature environments, and (3) class signature closures. Additionally, fields and methods, respectively, have (4) field signatures and (5) method signatures.

4.1 Class Signatures

If N is the set of all class names, and L is the set of all member (*i.e.*, field and method) names, we define a set S that includes all class signatures by the equation

$$\mathsf{S} = \mathsf{N} \times \mathsf{N}^* \times \mathsf{FS}^* \times \mathsf{MS}^* \tag{3}$$

where $\times$ and * are the cross-product and finite-sequences set constructors, respectively, $\mathsf{FS} = \mathsf{L} \times \mathsf{N}$ is the set of field signatures, and $\mathsf{MS} = \mathsf{L} \times \mathsf{N}^* \times \mathsf{N}$ is the set of method signatures.

The equation for S expresses that a *class signature* corresponding to a certain class is composed of four components:

(i) The class name (also used as a *signature name* for the class signature),

(ii) A finite sequence of names of *immediate supersignatures* of the signature, *i.e.*, of signatures corresponding to immediate superclasses of the class,

(iii) A finite sequence of field signatures corresponding to class fields, and

(iv) A finite sequence of method signatures corresponding to class methods.

The use of signature names (members of N) inside signatures characterizes class signatures as nominal constructs, where two signatures with different names but that are otherwise equal are different signatures.

The second component of a signature, a (possibly empty) sequence of signature names (*i.e.*, a member of N^*), is the *immediate supersignature names* component of the class signature. Having names of immediate supersignatures of a class signature explicitly included as a component of the class signature is an essential and critical feature in the modeling of nominal subtyping in nominally-typed OOP. Explicitly specifying the supersignatures of a class signature identifies the nominal structure of the class hierarchy immediately above the named class. This also agrees with the

inheritance of the contract associated with class names, which is a crucial semantic component of what is intended to be inherited in nominally-typed mainstream OOP.

The equation for field signatures expresses that a *field signature* is a pair of a field name (a member of L) and a class signature name. Similarly, the equation for method signatures expresses that a *method signature* is a triple of a method name, a sequence of class signature names (for the method parameters), and a signature name (for the method result).

Not all members of set S are class signatures. To agree with our intuitions about describing the interfaces of classes and their instances, a member s of S is a class signature if its supersignature names component, its field signatures component and its method signatures component (*i.e.*, the second, third and fourth components of s) have no duplicate signature names, field names, and method names, respectively (for simplicity, method overloading is not modeled in our model of OOP). It should be noted, however, that field names and method names are in separate name spaces and thus we allow a field and a method to have the same name.

Information in class signatures is derived from the text of classes of OO programs. Given that interfaces of objects are the basis for defining types in OO type systems, class signatures are the formal basis for nominally-typed OO type systems, so as to confirm that objects are used consistently and properly within a program (Ch. 2 of [5], and [7], give more details on types and typing in OOP).

4.2 Signature Environments

A *signature environment* is a finite set of class signatures that has unique class names, where each signature name is associated with exactly one class signature in the environment. [4] In addition to requiring the uniqueness of signature names, a finite set of class signatures needs to satisfy certain consistency conditions to function as a signature environment. A signature environment specifies two relations between signature names: an immediate supersignature relation and a direct-reference (adjacency) relation (The first relation is a subset of the second). These two relations can be represented as directed graphs. The consistency conditions on a signature environment constrain these two relations and their corresponding graphs.

As such, a finite set se of class signatures is a signature environment if and only if (i) A class signature, with the right signature name, belongs to se for each signature reference in each class signature of se, (ii) The graph for the supersignatures relation for se is an acyclic graph [5], and (iii) The set of field signatures and method signatures of each class signature s in se is a superset of the set of field signatures and method signatures of each supersignature named by the supersignatures component of s.

In agreement with inheritance in mainstream OO languages, the last condition makes class signatures in signature environments reflect the explicit inheritance

[4] Accordingly, function application notation can be used to refer to particular class signatures in a signature environment. If nm is a signature name guaranteed to be the name of some class signature in a signature environment se, we use function application notation, $se(nm)$, to refer to this particular class signature.

[5] This constraint forces any signature environment to have at least one class signature that has no supersignatures (*i.e.*, its second component is the empty sequence).

information in class-based OOP, by requiring a class signature to only extend (*i.e.*, add to) the set of members supported by an explicitly-specified supersignature. Requiring the members of a class signature to be a superset of the members of all of its supersignatures means that *exact* matching of member signatures is required. This requirement thus enforces an *invariant subtyping* rule for field and method signatures, mimicking the rule used in mainstream OO languages (such as Java and C#) before the addition of generics. This condition can be relaxed but we do not do so in this paper. More details are available in [5].

4.3 Signature Closures

Inside a class signature, class names can be viewed as "pointers" that refer to other class signatures. Without bindings of class names to corresponding class signatures, a single class signature that has name references to other class signatures is not a closed entity on its own. This motivates the notion of a signature closure. A closure of a class signature is a set of class signatures (a signature environment, in particular) that offers bindings to class names referred to in all elements of the set, such that the whole set has no "dangling pointers" in its references to other class signatures (*i.e.*, is referentially-closed) and has no redundant class signatures relative to some main class signature in the set (called the root class signature of the closure). A signature closure thus "closes" the root class signature by providing bindings for all class names referenced, directly or indirectly, in the signature. This motivates the following formal definition of signature closures.

A *signature closure* is a pair of a signature name and a signature environment. A pair $sc = (nm, se)$ of a signature name nm and a signature environment se is a signature closure if and only if there exists a class signature s in se with signature name nm and if the direct-reference (adjacency) relation corresponding to se is referentially-closed relative to s, and if this relation is the smallest such relation. Class signature s is then called the *root class signature* of sc. Relative to the root class signature, a signature environment is minimal, *i.e.*, contains no unnecessary class signatures. This minimality condition ensures that all class signatures in the signature environment of a signature closure are accessible via paths in the adjacency graph of the signature environment starting from (the node in the graph corresponding to) the root signature name, *i.e.*, that the signature environment has no redundant class signatures unnecessary for the root class signature.

Similar to a single class signature, when viewed as a "closed class signature" a signature closure has a name: namely, that of its root class signature; has member signatures: namely, field and method signatures of its root class signature; has a fields shape and a methods shape: namely, those of its root class signature; and it has immediate supersignature names: namely, those of its root class signature. A signature closure, not just a class signature, is the full formal expression of the notion of object interfaces (the fields shape is the set of field names, and similarly for methods. See [7] and Ch. 2 in [5] for more details on shapes and a discussion of object interfaces). Each class in a class-based OOP program has a corresponding class signature and a corresponding class signature closure. The nominal information in

a class signature closure is an invariant of all instances of the class (including the contracts associated with class names) .

4.4 Relations on Signatures

For class signatures $\mathsf{s}_1 = (\mathsf{nm}_1, \mathsf{nms}_1, \mathsf{fss}_1, \mathsf{mss}_1)$ and $\mathsf{s}_2 = (\mathsf{nm}_2, \mathsf{nms}_2, \mathsf{fss}_2, \mathsf{mss}_2)$, we define $\mathsf{s}_1 = \mathsf{s}_2 \Leftrightarrow (\mathsf{nm}_1 = \mathsf{nm}_2) \wedge (\mathsf{nms}_1 \equiv \mathsf{nms}_2) \wedge (\mathsf{fss}_1 \equiv \mathsf{fss}_2) \wedge (\mathsf{mss}_1 \equiv \mathsf{mss}_2)$ where $\equiv$ is an equivalence relation on sequences that ignores the order (and repetitions) of elements of a sequence. For two field signatures $\mathsf{fs}_1 = (\mathsf{a}_1, \mathsf{nm}_1)$ and $\mathsf{fs}_2 = (\mathsf{a}_2, \mathsf{nm}_2)$, $\mathsf{fs}_1 = \mathsf{fs}_2 \Leftrightarrow (\mathsf{a}_1 = \mathsf{a}_2) \wedge (\mathsf{nm}_1 = \mathsf{nm}_2)$. Similarly, for two method signatures $\mathsf{ms}_1 = (\mathsf{b}_1, \mathsf{nms}_1, \mathsf{nm}_1)$ and $\mathsf{ms}_2 = (\mathsf{b}_2, \mathsf{nms}_2, \mathsf{nm}_2)$, $\mathsf{ms}_1 = \mathsf{ms}_2 \Leftrightarrow (\mathsf{b}_1 = \mathsf{b}_2) \wedge (\mathsf{nms}_1 = \mathsf{nms}_2) \wedge (\mathsf{nm}_1 = \mathsf{nm}_2)$ (Here, sequence equality, not sequence equivalence, is used. For method parameter signature names, order and repetitions do matter.)

Two signature environments are equal if and only if they are equal as sets. Two signature closures are equal if and only if they are equal as pairs. Equal signature closures have the same root class signature name and equal signature environments.

Finally, a relation between signature environments that is needed when we discuss inheritance is the extension relation on signature environments. A signature environment se_2 extends a signature environment se_1 (written $se_2 \blacktriangleleft se_1$) if se_2 binds the names defined in se_1 to exactly the same class signatures as se_1 does. Viewed as sets, se_2 is a superset of se_1. Thus,

$$se_2 \blacktriangleleft se_1 \Leftrightarrow se_2 \supseteq se_1.$$

4.5 Subsigning and Inheritance

The supersignatures component of class signatures defines an ordering relation between signature closures. We call this relation between signature closures *subsigning*. The subsigning relation between class signature closures models the inheritance relation between classes in class-based OOP.

A signature closure $sc_2 = (nm_2, se_2)$ is an *immediate subsignature* ($\trianglelefteq_1$) of a signature closure $sc_1 = (nm_1, se_1)$ if the signature environment (*i.e.*, the second component) of sc_2 is an extension ($\blacktriangleleft$) of the signature environment of sc_1 and the signature name of sc_1 is a member of the supersignature names component of the root class signature of sc_2, *i.e.*,

$$sc_2 \trianglelefteq_1 sc_1 \Leftrightarrow se_2 \blacktriangleleft se_1 \wedge (nm_1 \in super\_sigs(se_2(nm_2))).$$

The subsigning relation, $\trianglelefteq$, between signature closures is the reflexive transitive closure of the immediate subsigning relation ($\trianglelefteq_1$). To illustrate the definitions given in this section, Appendix A presents a few examples of signature constructs, and presents an example of signature closures that are in the subsigning relation.

The inclusion of class contracts in deciding the subsigning relation makes the subsigning relation a more accurate reflection of a true "is-a" (substitutability) relationship than the structural subtyping relation used in structurally-typed OOP. This makes subsigning capture the fact that subtyping in nominally-typed OOP is more semantically accurate than structural subtyping (as explained in the discussion

in [7] and Ch. 2 of [5] of the Liskov Substitutability Principle and of semantic subtyping versus syntactic subtyping).

5 NOOP: A Model of Nominal OOP

Using the records domain constructor ($\multimap$) presented in Section 3 and signature constructs presented in Section 4, in this section we now present the construction of **NOOP** as a more precise model of nominally-typed mainstream OOP.

The construction of **NOOP** proceeds in two steps. First, the solution of a simple recursive domain equation defines a preliminary domain $\hat{\mathcal{O}}$ of raw objects, where an object in $\hat{\mathcal{O}}$ contains (1) a signature closure that encodes nominal information of nominally-typed OOP, and contains bindings for object members in two separate records: (2) a record for fields of the object, and (3) a record for methods of the object.

A simple recursive definition of objects with signature information does not force signature information embedded in objects to conform with their member bindings. Accordingly, in the second step of the construction of **NOOP**, invalid objects in the constructed preliminary domain of objects $\hat{\mathcal{O}}$ are "filtered out" producing a domain $\mathcal{O}$ of proper objects that model nominal objects of mainstream OO software. Invalid objects are ones where the signature information is inconsistent with member bindings in the member records. The filtering of the preliminary domain is done by defining a projection function on the preliminary domain $\hat{\mathcal{O}}$.

We call the model having the preliminary domain defined by the domain equation '*pre***NOOP**'. Our target model, **NOOP**, is the one containing the image domain resulting from applying the filtering function on the preliminary domain $\hat{\mathcal{O}}$ of *pre***NOOP**.

5.1 Construction of NOOP

The domain equation defining *pre***NOOP**, and thence **NOOP**, uses two flat domains $\mathcal{L}$ and $\mathcal{S}$. Domain $\mathcal{L}$ is the flat domain of labels, and domain $\mathcal{S}$ is the flat domain of signature closures (Section 4).

The domain equation that describes *pre***NOOP** is

$$\hat{\mathcal{O}} = \mathcal{S} \times (\mathcal{L} \multimap \hat{\mathcal{O}}) \times (\mathcal{L} \multimap (\hat{\mathcal{O}}^* \multimap\!\!\rightarrow \hat{\mathcal{O}})) \tag{4}$$

where the main domain defined by the equation, $\hat{\mathcal{O}}$, is the domain of raw objects, $\times$ is the strict product domain constructor, and $\multimap$ is the records domain constructor (Section 3). Equation (4) states that every raw object (*i.e.*, every element in $\hat{\mathcal{O}}$) is a triple of:

(i) A signature closure (*i.e.*, a member of $\mathcal{S}$),

(ii) A fields record (*i.e.*, a member of $\mathcal{L} \multimap \hat{\mathcal{O}}$), and

(iii) A methods record (*i.e.*, a member of $\mathcal{L} \multimap (\hat{\mathcal{O}}^* \multimap\!\!\rightarrow \hat{\mathcal{O}})$, where $\multimap\!\!\rightarrow$ is the strict continuous functions domain constructor, and * is the finite-sequences domain constructor).

Domain $\hat{\mathcal{O}}$ of *pre***NOOP** is the solution of Equation (4). Applying the iterative least-fixed point (LFP) construction method from domain theory [12], the construction of $\hat{\mathcal{O}}$ proceeds in iterations, driven by the structure of the right-hand side (RHS) of Equation (4). The RHS of the equation is viewed as a continuous function over domains (given the continuity of all used domain constructors, and that constructor composition preserves continuity.) Details of the iterative construction of *pre***NOOP** are presented in [5].

The second step in constructing **NOOP** is the definition of a projection/filtering function, `filter`, to map domain $\hat{\mathcal{O}}$ of *pre***NOOP** to the **NOOP** domain $\mathcal{O}$ of valid objects modeling objects of nominally-typed OOP. For this, first, we define an object in $\hat{\mathcal{O}}$ to be valid as follows.

Definition 5.1 An object o in $\hat{\mathcal{O}}$ is *valid* if it is the bottom object $\perp_{\mathcal{O}}$, or if it is a non-bottom object $o = (sc, fr, mr)$ such that

- The fields shape and the methods shape of sc are exactly the same as the shape of fr and the shape of mr, respectively,
- Non-bottom valid objects bound to field names in fr have signature closures that subsign the signature closures for corresponding fields in sc, and
- Non-bottom functions bound to method names in mr conform to corresponding method signatures in sc. By conformance the functions are required to take in sequences of valid objects whose embedded signature closures subsign (componentwise) the corresponding sequences of method parameter signature closures in sc, prepended with sc itself (for the implicit parameter `self/this`), and to return valid objects with signature closures that subsign the corresponding return value signature closures specified in the method signatures in sc.

The function `filter` mapping $\hat{\mathcal{O}}$ into $\hat{\mathcal{O}}$ ($\mathcal{O}$ is a proper subdomain of $\hat{\mathcal{O}}$) is defined using the following three recursive function definitions, presented using *lazy* functional language pseudo-code.

```
fun filter(o:Ô):O
  match o with ((nm,se), fr, mr)
  if (sf-shp(se(nm)) != rec-shp(fr)) ∨
     (sm-shp(se(nm)) != rec-shp(mr))
     return ⊥_O // non-matching shapes
  else // lazily construct closest valid object to o
     match se(nm), fr, mr with
       (_, _, [(a_i, snm_i) | i=1,···,m ],
              [(b_j, mi_snm_j, mo_snm_j) | j=1,···n]),
       (fr-tag, {a_i ↦ o_i | i=1,···,m}),
       (mr-tag, {b_j ↦ m_j | j=1,···,n})
     let si = se_clos(se, snm_i)
     let misj = map(se_clos(se), [nm::mi_snm_j])
         // nm is prepended to mi_snm_j to  handle 'this'
     let mosj = se_clos(se, mo_snm_j)
```

```
      return ((nm,se),
        (fr-tag, {a_i ↦ filter-obj-sig(si,o_i) | i=1,···,m}),
        (mr-tag, {b_j ↦ filter-meth-sig(misj, mosj, m_j)
                 | j=1,···,n}))

fun filter-obj-sig(ss:S, o:Ô):O
  match o with (s, _, _)
  if (s ⊴ ss)
     return filter(o) // closest valid object to o
  else
     return ⊥_O // no subsigning

fun filter-meth-sig(in_s:S⁺, out_s:S, m:M̂):M
  return (λos.let vos = map2(filter-obj-sig, in_s, os)
              in filter-obj-sig(out_s, m(vos)))
```

In the definition of `filter`, functions `sf-shp` and `sm-shp` compute field and method shapes of signatures, while function `rec-shp` computes shapes of records. Function `se_clos(se,nm)` computes a signature closure corresponding to signature name `nm` whose first component is `nm` and whose second component is the minimal subset of signature environment `se` that makes `se_clos(se,nm)` a signature closure. To handle `this/self` a "curried" version of `se_clos` is passed to the `map` function. Additionally, domain $\mathcal{S}^+$ is the domain of non-empty sequences of signature closures (non-empty because methods are always passed in at least the object `this/self`), and domains $\hat{\mathcal{M}}$ and $\mathcal{M}$ are auxiliary domains of raw methods and methods, respectively. The function `map2` is the two-dimensional version of `map` (*i.e.*, takes a binary function and two input lists as its arguments).

In words, the definition of the filtering function `filter` states that the function takes an object o of $\hat{\mathcal{O}}$ and returns a corresponding valid object of $\mathcal{O}$. If the object is invalid because of non-matching shapes in the signature closure of o and its member records, `filter` returns the bottom object $\bot_{\mathcal{O}}$ (in domain $\hat{\mathcal{O}}$, $\bot_{\mathcal{O}}$ is the closest valid object to an invalid object with non-equal shapes in its signature and records). Otherwise, o has matching signature and record shapes but may have objects bound to its fields, or taken in or returned by its methods, whose signature closure does not subsign the corresponding signature closures in the signature closure of o. In this case, `filter` lazily constructs and returns the closest valid object in domain $\hat{\mathcal{O}}$ to o, where all non-bottom fields and non-bottom methods of o are guaranteed (via functions `filter-obj-sig` and `filter-meth-sig`, respectively) to have signature closures that subsign the corresponding signature closures in the signature closure of o.

Function `filter-obj-sig` checks if its input object `o` has a signature closure `s` that subsigns a required declared signature closure `ss`. If `s` is not a subsignature of `ss`, `filter-obj-sig` returns $\bot_{\mathcal{O}}$. If it is, the function calls `filter` on `o`, thereby returning the closest valid object to `o`.

For methods, when `filter-meth-sig` is applied to a method `m` it returns a valid

method that when applied to the same input $os \in \hat{\mathcal{O}}^+$ as `m`, returns the closest valid object to the output object of `m` that subsigns the declared output signature closure `out_s` corresponding to the sequence of valid objects closest (component-wise) to *os* that (again, component-wise) subsigns the declared sequence of input signature closures `in_s` prepended with the signature closure of the object enclosing `m` (to properly filter the first argument object in *os*, which is the value for `this/self`). The proof that domain $\mathcal{O}$, as defined by `filter`, is a well-defined computable subdomain of $\hat{\mathcal{O}}$ is presented in [5].

5.2 Class Types

As constructed, **NOOP** is a nominal model of OOP, because objects of domain $\mathcal{O}$ of **NOOP** include signatures specifying the associated class contracts maintained by the objects (including inherited contracts). This nominal information encoded in signatures provides a framework for naturally partitioning the domain of **NOOP** objects into sets defining class types, where a type is a set of similar objects.

First, we define exact class types. The *exact class type* corresponding to a class `C` is the set of all objects tagged with the signature closure for `C`.[6] Next, it should be noted that a cardinal principle of nominally-typed mainstream OOP is that *objects from subclasses of a class* `C` *conform to the contract of class* `C` and can be used in place of objects constructed using class `C` (*i.e.*, in place of objects in the exact class type of `C`). Hence, the natural type associated with class `C`, called the *class type* corresponding to or designated by `C`, consists of the objects in class `C` plus the objects in *all* subclasses of class `C`. In nominally-typed OO languages, the class type designated by class `C` is not the exact class type for `C` but the union of all exact types corresponding to classes that subclass (*i.e.*, inherit from) class `C`, including class `C` itself.

Motivated by this discussion, we define class types in **NOOP** as interpretations of signature closures. For a signature closure *sc*, its interpretation $\mathbb{S}[sc]$ is a subdomain of domain $\mathcal{O}$, having the same underlying approximation ordering of domain $\mathcal{O}$, and whose universe is defined by the equation

$$|\mathbb{S}[sc]| = \{(scs, fr, mr) \in \mathcal{O} | scs \trianglelefteq sc\} \cup \{\bot_{\mathcal{O}}\}. \tag{5}$$

In other words, the class type designated by a class is the interpretation of the signature closure *sc* corresponding to the class, which, in turn, is the set of all objects in domain $\mathcal{O}$ of **NOOP** with a signature closure *scs* that subsigns *sc*, or the bottom object $\bot_{\mathcal{O}}$. Given that subsigning in **NOOP** models OO inheritance, the definition of **NOOP** class types is in full agreement with intuitions of mainstream OO developers.

5.3 Inheritance is *Subtyping*

Having constructed **NOOP**, and having defined class types based on intuitions of mainstream OO developers, we can now easily see what it means for nominally-typed

[6] In Java, for example, objects in the exact type for a class `C` are precisely those for which the `getClass()` method returns the class object for `C`.

mainstream OO type systems to completely identify inheritance and subtyping. We express this statement formally as follows: Two signature closures corresponding to two classes are in the subsigning relation if and only if the class types denoted by the two signature closures are in the subset relation (*i.e.*, the two classes are in the inheritance relation if and only if the corresponding class types are in the nominal subtyping relation). We prove the correspondence between inheritance and subtyping in the following theorem.

Theorem 5.2 *For two signature closures* sc_1 *and* sc_2 *denoting class types* $\mathbb{S}[sc_1]$ *and* $\mathbb{S}[sc_2]$*, we have*

$$sc_1 \trianglelefteq sc_2 \Leftrightarrow \mathbb{S}[sc_1] \subseteq \mathbb{S}[sc_2] \tag{6}$$

Proof Based on Equation (5), and the non-emptiness of class types [7], the proof of this theorem is simple.

Case: The $\Rightarrow$ (only if) direction:

If $sc_1 \trianglelefteq sc_2$, by applying the definition of $\mathbb{S}[sc_2]$ (Equation (5)) all elements of $\mathbb{S}[sc_1]$ belong to $\mathbb{S}[sc_2]$ (the variable *scs* in Equation (5) is instantiated to sc_1, and $\bot_{\mathcal{O}}$ is a common member in all class types). Thus, $\mathbb{S}[sc_1] \subseteq \mathbb{S}[sc_2]$.

Case: The $\Leftarrow$ (if) direction:

By the non-emptiness of $\mathbb{S}[sc_1]$ there exists a non-bottom object o of $\mathbb{S}[sc_1]$ with signature closure sc_1. If $\mathbb{S}[sc_1] \subseteq \mathbb{S}[sc_2]$, then $o \in \mathbb{S}[sc_2]$. By Equation (5) all non-bottom members of $\mathbb{S}[sc_2]$ must have a signature closure that subsigns sc_2. When applied to o we thus have $sc_1 \trianglelefteq sc_2$. □

We should notice in the proof above that it is the nominality of objects of **NOOP** (*i.e.*, the embedding of signature closures into objects) that makes $\mathbb{S}[sc_2]$ being a superset of $\mathbb{S}[sc_1]$ imply that sc_1 has sc_2 as one of its supersignatures, and vice versa. The simplicity of the proof is a clear indication of the naturalness of the definitions for class signatures and class types.

6 Conclusions

In this paper we presented **NOOP** as a model of OOP that includes nominal information found in nominally-typed mainstream OO software. This led us to readily prove that inheritance, at the syntactic level, and subtyping, at the semantic level, completely agree in nominally-typed OOP. It is necessary, we thus believe, to include nominal information in any accurate model of nominally-typed mainstream OOP. By its inclusion of nominal information, **NOOP** offers a chance to understand and advance OOP and current OO languages based on a firmer semantic foundation.

[7] A class type $\mathbb{S}[sc]$ is always non-empty (*i.e.*, always has some non-bottom object) because the object $(sc, \{a_1 \mapsto \bot_{\mathcal{O}}, \cdots, a_m \mapsto \bot_{\mathcal{O}}\}, \{b_1 \mapsto \bot_{\mathcal{M}}, \cdots, b_n \mapsto \bot_{\mathcal{M}}\})$ (where $\{a_1, \cdots, a_m\}$ is the fields shape of sc and $\{b_1, \cdots, b_n\}$ is the methods shape of sc) is always a valid constructed object (*i.e.*, is an object of domain $\hat{\mathcal{O}}$ of *pre***NOOP** that passes filtering to domain $\mathcal{O}$ of **NOOP**). This object is a member of $\mathbb{S}[sc]$ by Equation (5).

7 Future Work

The possibilities for research that can be built on top of research presented in this paper are many. One immediate possible future work is to define a minimal nominally-typed OO language, *e.g.*, in the spirit of FJ [20], then, in a standard straightforward manner, give the denotational semantics of program constructs of this language in **NOOP**. The type safety of this language can then be proven using the given denotational semantics.

Another possible future work that can be built on top of **NOOP** is to produce a denotational model of *generic* nominally-typed OOP. Such a model may provide a chance for a better analysis of features of generics in nominally-typed mainstream OO languages and thus provide a chance for suggesting improvements and extensions to the type systems of these languages.

Acknowledgement

The author would like to much thank Professor Robert "Corky" Cartwright for the discussions we had and the guidance he gave that helped in developing **NOOP** and in reaching some of the conclusions in this paper, as well as to thank Professor Benjamin Pierce for the feedback he offered on motivating and presenting **NOOP**.

The author would also like to dedicate this work to his late beloved mother. She has been a very motivating soul and was a major supporter of him during the development of research presented in this paper. Mum, I miss you much.

References

[1] C# language specification, version 3.0. http://msdn.microsoft.com/vcsharp, 2007.

[2] *ISO/IEC 14882:2011: Programming Languages: C++*. 2011.

[3] Martin Abadi and Luca Cardelli. A semantics of object types. In *Proc. LICS'94*, 1994.

[4] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[5] Moez A. AbdelGawad. *NOOP: A Mathematical Model of Object-Oriented Programming*. PhD thesis, Rice University, 2012.

[6] Moez A. AbdelGawad. In nominally-typed object-oriented programming objects are *Not* merely records and inheritance *Is* subtyping. *Submitted for publication*, 2013.

[7] Moez A. AbdelGawad. An overview of nominal-typing versus structural-typing in object-oriented programming. Technical report, arXiv.org:1309.2348 [cs.PL], 2013.

[8] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[9] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the internat. symp. on semantics of data types*, volume 173, pages 51–67. Springer-Verlag, 1984.

[10] Luca Cardelli. A semantics of multiple inheritance. *Inform. and Comput.*, 76:138–164, 1988.

[11] Robert Cartwright and Moez A. AbdelGawad. Inheritance *Is* subtyping (extended abstract). In *The 25^{th} Nordic Workshop on Programming Theory (NWPT)*, Tallinn, Estonia, 2013.

[12] Robert Cartwright and Rebecca Parsons. Domain theory: An introduction, 1988. Monograph (based on earlier notes by Dana Scott).

[13] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown Univ., 1989.

[14] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *POPL'90 Proceedings*, 1990.

[15] William R. Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *ACM Symposium on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 433–444, 1989.

[16] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. W. Mislove, and D. S. Scott. *Continuous Lattices and Domains*, volume 93 of *Encyclopedia Of Mathematics And Its Applications*. Cambridge University Press, 2003.

[17] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Springer-Verlag, 1978.

[18] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2005.

[19] C. A. Gunter and Dana S. Scott. *Handbook of Theoretical Computer Science*, chapter Semantic Domains. 1990.

[20] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[21] John McCarthy. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, pages 33–70, 1963.

[22] John McCarthy. Towards a mathematical science of computation. *Science*, 1996.

[23] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[24] Martin Odersky. The scala language specification, v. 2.7. http://www.scala-lang.org, 2009.

[25] Gordon D. Plotkin. Domains. Lecture notes in advanced domain theory, 1983.

[26] Bernhard Reus. Modular semantics and logics of classes. volume 2803, pages 456–469. Springer-Verlag, 2003.

[27] Bernhard Reus and Thomas Streicher. Semantics and logic of object calculi. 2002.

[28] Bernhard Reus and Thomas Streicher. Semantics and logics of objects. *Proceedings of the 17th Symp. on Logic in Computer Science (LICS 2002)*, pages 113–122, 2002.

[29] Dana S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–587, 1976.

[30] Dana S. Scott. Domains for denotational semantics. Technical report, Computer Science Department, Carnegie Mellon University, 1983.

[31] Anthony J. H. Simons. The theory of classification, part 1: Perspectives on type compatibility. *Journal of Object Technology*, 1(1):55–61, May-June 2002.

[32] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

A Signature Examples

To illustrate the definitions of signature constructs given in Section 4, in this appendix we present a few examples of signature constructs. Assuming the following OO class definitions (in Java-like pseudo-code),

```
class Object {
  Boolean equals(Object o){ ... }
}
```

```
class Boolean extends Object {
  Boolean equals(Object b){ ... }
  ... // other members of class Boolean
}

class Pair extends Object {
  Object first, second;
  Boolean equals(Object p){ ... }
  Pair swap(){ return new Pair(second, first); }
}
```

we define the corresponding class signatures

$ObjSig$ =(Object, [], [], [(equals, [Object], Boolean)]),

$BoolSig$ =(Boolean, [Object], ...), and

$PairSig$ =(Pair, [Object], [(first, Object), (second, Object)],
[(equals, [Object], Boolean), (swap, [], Pair)])

and, hence, define signature environments $ObjSigEnv$ = {$ObjSig$, $BoolSig$}, and $PairSigEnv$ = {$ObjSig$, $BoolSig$, $PairSig$}, and the signature closures $ObjSigClos$ =(Object, $ObjSigEnv$), and $PairSigClos$ = (Pair, $PairSigEnv$).

We can immediately see, using the definition of extension and the definitions of immediate subsigning and subsigning in Section 4, that $PairSigEnv \blacktriangleleft ObjSigEnv$, $PairSigClos \trianglelefteq_1 ObjSigClos$, and $PairSigClos \trianglelefteq ObjSigClos$. The last conclusion expresses the fact that class Pair inherits from class Object, and the second to last conclusion expresses that class Pair is an immediate subclass of class Object (The reader is encouraged to find other similar conclusions based on the definitions of classes Object, Boolean and Pair given above.)